Elements of a Distributed Programming System

5-JAN-76

James E. White
Augmentation Research Center

Stanford Research Institute
Menlo Park, California 94025

(415) 326-6200 x2960

This paper suggests some extensions to the simple Procedure Call Protocol described in a previous paper (27197,). By expanding the procedure call model and standardizing other common forms of inter-process interaction, such extensions would provide the applications programmer with an even more powerful distributed programming system.

The work reported here was supported by the Advanced Research Projects Agency of the Department of Defense, and by the Rome Air Development Center of the Air Force.

This paper will be submitted for publication in the Journal of Computer Languages.

INTRODUCTION 1

In a companion paper [1], the author proposes a simple protocol and software framework that would facilitate the construction of distributed systems within a resource-sharing computer network by enabling distant processes to communicate with one another at the procedure call level. Although of great utility even in its present form, this rudimentary "distributed programming system (DPS)" supports only the most fundamental aspects of remote procedure calling. In particular, it permits the caller to identify the remote procedure to be called, supply the necessary arguments, determine the outcome of the procedure, and recover its results. The present paper extends this simple procedure call model and standardizes other common forms of process interaction to provide a more powerful and comprehensive distributed programming system. The particular extensions proposed in this paper serve hopefully to reveal the DPS concept's potential, and are offered not as dogma but rather as stimulus for further research. 1a

The first section of this paper summarizes the basic distributed programming system derived in [1]. The second section describes the general strategy to be followed in extending it. The third and longest section identifies and explores some of the aspects of process interaction that are sufficiently common to warrant standardization, and suggests methods for incorporating them in the DPS model. 1b

REVIEWING THE BASIC SYSTEM 2

The distributed programming system derived in [1] assumes the existence of and is built upon a network-wide "inter-process communication (IPC)" facility. As depicted in Figure 1, DPS consists of a high-level model of computer processes and a simple, application-independent "procedure call protocol (PCP)" that implements the model by regulating the dialog between two processes interconnected by means of an IPC communication "channel". DPS is implemented by an installation-provided "run-time environment (RTE)", which is link loaded with (or otherwise made available to) each applications program. 2a

The Model 2b

The procedure call model (hereafter termed the Model) views a process as a collection of remotely callable subroutines or "procedures". Each procedure is invoked by name, can be supplied a list of arguments, and returns to its caller both a boolean outcome, indicating whether it succeeded or failed, and a list of results. The Model permits the process at either end of the IPC channel to invoke procedures in its neighbor, and further permits a process to accept two or more procedure calls for concurrent execution. 2b1

The arguments and results of procedures are modeled from a small set of primitive "data types", listed below: 2b2

LIST: A list is an ordered sequence of N data objects called "elements" (here and throughout these descriptions, N is confined to the range [0, 2**15-1]). A LIST may contain other LISTs as elements, and can therefore be employed to construct arbitrarily complex, composite arguments or results.

CHARSTR: A character string is an ordered sequence of N ASCII characters, and conveniently models a variety of textual entities, from short user names to whole paragraphs of text.

BITSTR: A bit string is an ordered sequence of N bits and, therefore, provides a means for representing arbitrary binary data (for example, the contents of a word of memory).

INTEGER: An integer is a fixed-point number in the range [-2**31, 2**31-1], and conveniently models various kinds of numerical data, including time intervals, distances, and so on.

INDEX: An index is an integer in the range [1, 2**15-1]. As its name and value range suggest, an INDEX can be used to address a particular bit or character within a string, or element within a list. Furthermore, many of the protocol extensions to be proposed in this paper will employ INDEXes as handles for objects within the DPS environment (for example, processes and channels).

BOOLEAN: A boolean represents a single bit of information, and has either the value true or false.

EMPTY: An empty is a valueless place holder within a LIST or parameter list.

The Protocol 2c

The procedure call protocol (hereafter termed the Protocol), which implements the Model, defines a "transmission format" (like those suggested in Appendix A) for each of the seven data types listed above, and requires that parameters be encoded in that format whenever they are transported between processes. 2c1

The Protocol also specifies the inter-process messages by which remote procedures are invoked. These messages can be described symbolically as follows: 2c2

```
   message-type=CALL    [tid] procedure-name arguments
   message-type=RETURN   tid  outcome        results
```

The first message invokes the procedure whose NAME is specified using the ARGUMENTS provided. The second is returned in eventual response to the first and reports the OUTCOME and RESULTS of the completed procedure. Whenever OUTCOME indicates that a procedure has failed, the procedure's RESULTS are required to be an error number and diagnostic message, the former to help the invoking program determine what to do next, the latter for possible presentation to the user. The presence of an optional "transaction identifier (TID)" in the CALL message constitutes a request by the caller for an acknowledging RETURN message echoing the identifier. 2c3

Although data types and their transmission formats serve primarily as vehicles for representing the arguments and results of remote procedures, they can just as readily and effectively be employed to represent the messages by which those parameters are transmitted. The Protocol, therefore, represents each of the two messages described above as a PCP data object, namely, a LIST whose first element is an INDEX message type. The following concise statement of the Protocol results: 2c4

```
   LIST (CALL,   tid,     procedure, arguments)
         INDEX=1 [INDEX]  CHARSTR    LIST
   LIST (RETURN, tid,     outcome,   results)
         INDEX=2 INDEX    BOOLEAN    LIST
```

Here and in subsequent protocol descriptions, elements enclosed in square brackets are optional (that is, may be EMPTY). The RESULTS of an unsuccessful procedure would be represented as follows: 2c5

```
   LIST (error, diagnostic)
         INDEX  CHARSTR
```

The Run-Time Environment 2d

The run-time environment (hereafter termed the environment) interfaces the applications program to a remote process via an IPC channel. In doing so, it provides the applications program with a collection of "primitives", implemented either as subroutines or system calls, that the applications program can employ to manipulate the remote process to which the channel connects it. The environment implements these primitives by sending and receiving various protocol messages via the channel. 2d1

In its present rudimentary form, the Protocol enables the environment to make a single, remote procedure calling primitive like the following available to the applications program: 2d2

```
CALLPROCEDURE (procedure, arguments -> outcome, results)
               CHARSTR    LIST         BOOLEAN  LIST
```

This primitive invokes the indicated remote PROCEDURE using the ARGUMENTS provided and returns its OUTCOME and RESULTS. While this primitive blocks the invoking applications program until the remote procedure returns, a variant that simply initiates the call and allows the applications program to collect the outcome and results in a second operation can also be provided. 2d3

Since the interface between the environment and the applications program is machine- and possibly even language-dependent, environment-provided primitives can only be described in this paper symbolically. Although PCP data types provide a convenient vehicle for describing their arguments and results and are therefore used for that purpose above and throughout the paper, such parameters will normally be transmitted between the environment and the applications program in some internal format. 2d4

BOOTSTRAPPING THE NEW PROTOCOL FUNCTIONS 3

Since the Protocol already provides a mechanism for invoking arbitrary remote procedures, the Model extensions to be proposed in this paper will be implemented whenever possible as procedures, rather than as additional messages. Unlike applications procedures, these special "system procedures" will be called and implemented by run-time environments, rather than by the applications programs they serve. Although inaccessible to the remote applications program via the normal environment-provided remote procedure calling primitive, system procedures will enable the environment to implement and offer new primitives to its applications program. 3a

The calling sequences of many of these new primitives will closely correspond to those of the remote system procedures by which they are implemented. Other primitives will be more complex and require for their implementation calls to several system procedures, possibly in different processes. Besides describing the Protocol additions required by the various Model extensions proposed, the author will, throughout this paper, suggest calling sequences for the new primitives that become available to the applications program. 3b

SOME POSSIBLE EXTENSIONS TO THE MODEL 4

1. Creating Remote Processes 4a

Before a program in one machine can use resources in another, it must either create a new process in the remote machine, or gain access to an existing one. In either case, the local process must establish an IPC channel to a resident dispatching process within the remote system, specify the program to be started or contacted, and identify itself so that its access to the program can be established and billing carried out. After these preliminary steps have been accomplished, the requested process assumes responsibility for the IPC channel and substantive communication begins. 4a1

The manner in which the environment carries out the above scenario is largely dictated by the IPC facility upon which the distributed system is based. If the IPC facility itself provides a single primitive that accomplishes the entire task, then the environment need only invoke that primitive. If, on the other hand, it only provides a mechanism by which the environment can establish a channel to the remote dispatcher, as is the case within the ARPA Computer Network (the ARPANET), then the Protocol itself must contain provisions for naming the program to be run and presenting the required credentials. 4a2

Adding to the Protocol the following system procedure enables the local environment to provide the remote dispatcher with the necessary information in this latter case: 4a3

```
   INIPROCESS (program, credentials)
               CHARSTR  LIST (user,   password, account)
                              CHARSTR CHARSTR   CHARSTR
```

Its arguments include the name of the applications PROGRAM to be run; and the USER name, PASSWORD, and ACCOUNT of the local user to whom its use is to be billed. 4a4

This new procedure effectively adds to the Model the notion of "process creation", and enables the environment to offer the following primitives to its applications program: 4a5

```
   CRTPROCESS (computer, program, credentials -> ph)
               CHARSTR   CHARSTR  (as above)      INDEX
   DELPROCESS (ph)
               INDEX
```

The first primitive creates a new process or establishes contact with an existing one by first creating a channel to the dispatcher within the indicated COMPUTER and then invoking the remote system procedure INIPROCESS with the specified PROGRAM name and CREDENTIALS as arguments. The primitive returns a "process handle PH" by which the applications program can refer to the newly created process in subsequent dialog with the local environment. The process handle may be a token returned to the environment by the IPC facility, an index into a table within the environment, or anything else the environment's implementer may find convenient. 4a6

The second primitive "deletes" the previously created process whose handle PH is specified by simply deleting the IPC channel to the remote process and reclaiming any internal table space that may have been allocated to the process. 4a7

2. Introducing Processes to One Another 4b

The simplest distributed systems begin with a single process that creates, via the CRTPROCESS primitive described above, one or more "inferior" processes whose resources it requires. Some or all of these inferiors may in turn require other remote resources and so create inferiors of their own. This creative activity can proceed, in principle, to arbitrary depth. The distributed system is thus a tree structure whose nodes are processes and whose branches are IPC channels. 4b1

Although a distributed system can include an arbitrarily large number of processes, each process is cognizant of only the process that created it and those it itself creates, that is, its parent and sons. The radius within which a process can access the resources of the tree is thus artificially small. This limited sharing range, which prevents the convenient implementation of many distributed systems, can be overcome by extending the Model to permit an arbitrarily complex network of communication paths to be superimposed upon the process tree. 4b2

One of the many ways by which the Protocol can provide for such communication paths is to permit one process to "introduce" and thereby make known to one another any two processes it itself knows (for example, two of its sons, or its parent and son). Once introduced, the two processes would be able to invoke one another's procedures with the same freedom the introducing process enjoys. They could also introduce one another to other processes, and so create even longer communication paths. 4b3

2.1 Introductions Within a Homogeneous Environment 4b4

Provided one remains within a "homogeneous environment" (that is, the domain of a single IPC facility), the introduction of two processes requires little more than the formation of an IPC channel between them. Adding to the Protocol the following system procedures, which manipulate IPC "ports", enables the run-time environment of the process performing the introduction to negotiate such a channel:

```
ALOPORT (-> ph,   computer, port)
            INDEX CHARSTR   any
CNNPORT (ph,   computer, port)
         INDEX CHARSTR   any
DCNPORT (ph)
         INDEX
```

The detailed calling sequences for these procedures are dictated by the IPC facility that underlies the distributed system. Those above are therefore only representative of what may be required within any particular network, but are only slightly less complicated than those required, for example, within the ARPANET.

To create the channel, the introducing process' run-time environment allocates a PORT in each target process via ALOPORT, and then instructs each process via CNNPORT to connect its port to the other's via the IPC facility. The process handle PH returned by ALOPORT serves as a handle both initially for the allocated port, and then later for the process to which the attached channel provides access. To "separate" the two processes, the introducing process' environment need only invoke the DCNPORT procedure in each process, thereby dissolving the channel, releasing the associated ports, and deallocating the process handles.

Armed with these three new system procedures, the environment can provide the following new primitives to its applications program:

```
ITDPROCESS (ph1,  ph2 -> ph12, ph21, ih)
            INDEX INDEX  INDEX INDEX INDEX
SEPPROCESS (ih)
            INDEX
```

The first primitive introduces the two processes whose handles PH1 and PH2 are specified. Each handle may designate either a son, in which case the handle is one returned by CRTPROCESS; the parent process, for which a special handle (for example, 1) must always be defined; or a previously introduced process, in which case the handle is one obtained in a previous invocation of ITDPROCESS.

ITDPROCESS returns handles PH12 and PH21 by which the two processes will know one another, as well as an "introduction handle IH" that the applications program can later employ to separate the two processes via SEPPROCESS. The applications program initiating the introduction assumes responsibility for communicating to each introduced applications program its handle for the other.

2.2 Introductions within a Heterogeneous Environment 4b5

While their interconnection via an IPC channel is sufficient to introduce two processes to one another, in a heterogeneous environment the creation of such a channel is impossible. Suppose, as depicted in Figure 2, that processes P1 and P2 (in computers C1 and C2, respectively) are interconnected within a distributed system by means of a network IPC facility. Assume further that P2 attaches to the system another process P3 in a minicomputer M that although attached to C2 is not formally a part of the network. With this configuration, it is impossible for P2 to introduce processes P1 and P3 to one another by simply establishing an IPC channel between them, since they are not within the domain of a single IPC facility.

One way of overcoming this problem is to extend the Model to embrace the notion of a composite or "logical channel" composed of two or more physical (that is, IPC) channels. A message transmitted by process P1 via the logical channel to Pn (n=3 in the example above) would be relayed over successive physical channels by the environments of intermediate processes P2 through Pn-1. Although more expensive than physical channels, since each message must traverse at least two physical channels and be handled by all the environments along the way, logical channels would nevertheless enable processes that could not otherwise do so to access one another's resources. Since the relaying of messages is a responsibility of the environment, the applications program need never be aware of it.

As depicted in Figure 3, a logical channel would consist of table entries maintained by the environment of each process P1 through Pn, plus the committment of each environment to forward messages that arrive with a "routing code" addressing the local table entry. Each table entry would contain process handles for the two adjacent processes, as well as the routing code recognized by each. To communicate a message to its distant neighbor, the source process (say P1) would transmit it via its IPC channel to P2, with a routing code addressing the appropriate table entry within P2. Upon receipt of the message, P2 would locate its table entry via the routing code, update the message with the routing code recognized by P3, and forward the message to P3. Eventually the message would reach its final destination, Pn.

Adding to the Protocol the following system procedures enables the environment to construct a logical channel like that described above:

```
CRTROUTE (mycode, oldcode -> code, ph)
           INDEX   [INDEX]     INDEX INDEX
DELROUTE (yourcode)
           INDEX
```

The simplest logical channel (n=3) is created by P2, which invokes CRTROUTE in both P1 and P3, specifying in each case the routing code MYCODE it has assigned to its segment of the logical channel, and receiving in return the routing CODEs and process handles PHs assigned by the two processes. OLDCODE is not required in this simple case and is therefore EMPTY.

More complicated logical channels (n>3) are required when one or both of the processes to be introduced is already linked, by a logical channel, to the process performing the introduction. In such cases, a portion of the new channel to be constructed must replicate the existing channel, and hence the routing code OLDCODE for the table entry that represents that channel within the target process is specified as an additional argument of the system procedure. The target process must call CRTROUTE recursively in the adjacent process to replicate the rest of the model channel.

The process Pi that creates a logical channel assumes responsibility for insuring that it is eventually dismantled. It deletes the logical channel by invoking DELROUTE in Pi-1 and Pi+1, each of which propagates the call toward its end of the channel.

3. Controlling Access to Local Resources 4c

The process introduction primitive proposed above effectively permits access to a process to be transmitted from one process to another. Any process P2 that already possesses a handle to a process P1 can obtain a handle for use by a third process P3. Once P1 and P3 have been introduced, P3 can freely call procedures in P1 (and vice versa). 4c1

Although a process can, by aborting the ALOPORT system procedure, prevent its introduction to another process and so restrict the set of processes that gain access to it, finer access controls may sometimes be required. A process may, for example, house two separate resources, one of which is to be made available only to its parent (for example), and the other to any process to which the parent introduces it. Before such a strategy can be conveniently implemented, the Model must be extended to permit access controls to be independently applied to individual resources within a single process. 4c2

Although a single procedure can be considered a resource, it is usually more practical and convenient to conceive of larger, composite resources consisting of a number of related procedures. A simple data base management module containing procedures for creating, deleting, assigning values to, reading, and searching for data objects exemplifies such composite resources. Although each procedure is useless in isolation, the whole family of procedures provides a meaningful service. Such "packages" of logically related procedures might thus be the most reasonable object of the finer access controls to be defined. 4c3

Access controls can be applied to packages by requiring that a process first "open" and obtain a handle for a remote package before it may call any of the procedures it contains. When the process attempts to open the package, its right to do so can be verified and the attempt aborted if necessary. Challenging the open attempt would, of course, be less expensive than challenging every procedure call. The opening of a package would also provide a convenient time for package-dependent state information to be initialized. 4c4

Adding to the Protocol the following pair of system procedures enables the environment to open and close packages within another process. For efficiency, these procedures manipulate an arbitrary number of packages in a single transaction: 4c5

```
OPNPACKAGE (packages ->   pkhs)
            LISTofCHARSTRs LISTofINDEXs
CLSPACKAGE (pkhs)
            (as above)
```

The first procedure opens and returns "package handles PKHS" for the specified PACKAGES; the second closes one or more packages and releases the handles PKHS previously obtained for them. 4c6

Besides incorporating these two new system procedures, the Protocol must further require that a package handle accompany the procedure name in every CALL message (an EMPTY handle perhaps designating a system procedure). Note that this requirement has the side effect of making the package the domain within which procedure names must be unique. 4c7

The system procedures described above enable the environment to make available to its applications program, primitives that have calling sequences similar to those of the corresponding system procedures but which accept the process handle of the target process as an additional argument. Their implementation requires only that the environment identify the remote process from its internal tables and invoke OPNPACKAGE or CLSPACKAGE in that process. 4c8

4. Standardizing Access to Global Variables 4d

Conventional systems often maintain global "variables" that can be accessed by modules throughout the system. Such variables are typically manipulated using primitives of the form: 4d1

(1) Return the current value of V.
(2) Replace the current contents of V with a new value.

These primitives are either provided as language constructs or implemented by specialized procedures. The former approach encourages uniform treatment of all variables within the system. 4d2

Those distributed systems that maintain remotely-accessible variables must also select a strategy for implementing the required access primitives. While such primitives can, of course, be

implemented as specialized applications procedures, adding to the Protocol the following new system procedures insures a uniform run-time access mechanism: 4d3

```
   RDVARIABLE (pkh,  variable -> value)
                INDEX CHARSTR       any
   WRVARIABLE (pkh,  variable, value)
                INDEX CHARSTR    any
```

These procedures effectively define variables as named data objects modeled from PCP data types, and suggest that they be clustered in packages with related procedures. The system procedures return and specify, respectively, the VALUE of the VARIABLE whose name and package handle PKH are specified. 4d4

These new procedures enable the environment to make available to its applications program, primitives that have calling sequences similar to those of the corresponding system procedures but which accept the process handle of the target process as an additional argument. These primitives provide a basis upon which a suitably modified compiler can reestablish the compile-time uniformity that characterizes the manipulation of variables in conventional programming environments. Their implementation requires only that the local environment identify the remote process from its internal tables and invoke RDVARIABLE or WRVARIABLE in that process. 4d5

Most variables will restrict the range of data types and values that may be assigned to them; some may even be read-only. But because they are modeled using PCP data types, their values can, in principle, be arbitrarily complex (for example, a LIST of LISTs) and the programmer may sometimes wish to manipulate only a single element of the variable (or, if that element is itself a LIST, just one of its elements; and so on, to arbitrary depth). 4d6

Adding the following argument to their calling sequences extends the system procedures proposed above to optionally manipulate a single element of a variable's composite value: 4d7

```
   substructure
   [LISTofINDEXs]
```

At successive levels of the value's tree structure, the INDEX of the desired element is identified; the resulting list of indices identifies the SUBSTRUCTURE whose value is to be returned or replaced. 4d8

5. Routing Parameters Between Procedures 4e

In conventional programming systems, the results of procedures are used in a variety of ways, depending upon the context of the calls made upon them. A result may, for example: 4e1

(1) Provide the basis for a branch decision within the calling program.
(2) Become an argument to a subsequent procedure call.
(3) Be ignored and thus effectively discarded.

At run-time, the knowledge of a result's intended use usually lies solely within the calling program, which examines the result, passes it to a second procedure, or ignores it as it chooses. 4e2

In a distributed system, the transportation of results from callee to caller, carried out by means of one or more inter-process messages, can be an expensive operation, especially when the results are large. Data movement can be reduced in Cases 2 and 3 above by extending the Model to permit the intended disposition of each procedure result to be made known in advance to the callee's environment. In Case 2, provided both callees reside within the same process, the result can be held at its source and later locally supplied to the next procedure. In Case 3, the result can be discarded at its source (perhaps not even computed), rather than sent and discarded at its destination. 4e3

5.1 Specifying Parameters Indirectly 4e4

Variables offer potential for eliminating the inefficiencies involved in Case 2 above by providing a place within the callees' process where results generated by one procedure can be held until required by another. The Protocol can be extended to permit variables to be used in this way by allowing the caller of any procedure to include optional "argument- and result-list masks" like the following as additional parameters of the CALL message:

```
parameter list mask
[LIST (variable, ...)]
        [CHARSTR]
```

A parameter list mask would permit each parameter to be transmitted either directly, via the parameter list, or indirectly via a VARIABLE within the callee's process. Thus each

element of the mask specifies how the callee's environment is to obtain or dispose of the corresponding parameter. To supply the result of one procedure as an argument to another, the caller need only then appropriately set corresponding elements of the result and argument list masks in the first and second calls, respectively. The result list mask should be ignored if the procedure fails, and the error number and diagnostic message returned directly to the caller.

5.2 Providing Scratch Variables For Parameter Routing 4e5

Although each applications program could provide variables for use as described above, a more economical approach is to extend the Model to permit special "scratch variables", maintained by the environment without assistance from its applications program, to be created and deleted as necessary at run-time. Adding to the Protocol the following pair of system procedures enables the local environment to create and delete such variables in a remote process:

```
CRTVARIABLE (variable, value)
             CHARSTR   any
DELVARIABLE (variable)
             CHARSTR
```

These procedures create and delete the specified VARIABLE, respectively. CRTVARIABLE also assigns an initial VALUE to the newly-created variable.

These new procedures enable the environment to make available to its applications program, primitives that have calling sequences similar to those of the corresponding system procedures but which accept the process handle of the target process as an additional argument. Their implementation requires only that the environment identify the remote process from its internal tables and invoke CRTVARIABLE or DELVARIABLE in that process.

5.3 Discarding Results 4e6

The inefficiencies that result in Case 3 above are conveniently eliminated by allowing the caller to identify via the result list mask (for example, via a zero-length CHARSTR) that a result will be ignored and therefore need not be returned to the caller.

6. Supporting a Richer Spectrum of Control Transfers 4f

As currently defined by the Model, a procedure call is a simple, two-stage dialog in which the caller first describes the operation it wishes performed and the callee, after performing the operation, reports its outcome. Although this simple dialog form is sufficient to conveniently implement a large class of distributed systems, more complex forms are sometimes required. The Model can be extended to admit a variety of more powerful dialog forms, of which the four described below are examples. 4f1

6.1 Transferring Control Between Caller and Callee 4f2

Many conventional programming systems permit caller and callee to exchange control any number of times before the callee returns. Such "coroutine linkages" provide a means, for example, by which the callee can obtain help with a problem that it has encountered or deliver the results of one sub-operation and obtain the arguments for the next.

Adding to the Protocol the following system procedure, whose invocation relinquishes control of another, previously initiated procedure, enables the environment to effect a coroutine linkage between caller and callee:

```
TAKEPROCEDURE (tid,  yourtid, parameters)
               INDEX BOOLEAN  LIST
```

Its arguments include the identifier TID of the affected transaction, an indication YOURTID of from whose name space the identifier was assigned (that is, whether the process relinquishing control is the caller or callee), and PARAMETERS provided by the procedure surrendering control. By exploiting an existing provision of the Protocol (that is, by declining acknowledgment of its calls to TAKEPROCEDURE,) the invoking environment can effect the control transfer with a single inter-process message.

The addition of this new procedure to the Protocol enables the environment to provide the following new primitive to its applications program:

```
LINKPROCEDURE (tid,  arguments -> outcome,  results)
               INDEX LIST         [BOOLEAN] LIST
```

This primitive assumes that the CALLPROCEDURE primitive is also modified to return the pertinent transaction identifier should the callee initiate a coroutine linkage rather than return. Invocation of LINKPROCEDURE then continues the dialog by supplying ARGUMENTS and returning control to the remote procedure, and then awaiting the next transfer of control and the RESULTS that accompany it. If the remote procedure then returns, rather than initiating another coroutine linkage, the primitive reports its OUTCOME and invalidates the transaction identifier.

While this primitive blocks the applications program until the remote procedure relinquishes control, a variant that simply initiates the coroutine linkage and allows the applications program to collect the outcome and results in a second operation can also be provided.

6.2 Signalling the Caller/Callee 4f3

A monolog is often more appropriate than the dialog initiated by a coroutine linkage. The caller or callee might wish, for example, to report an event it has detected or send large parameters piecemeal to minimize buffering requirements. Since no return parameters are required in such cases, the initiating procedure need only "signal" its partner, while retaining control of the call.

Adding to the Protocol the following system procedure extends the Model to support signals and enables the environment to transmit parameters to or from another, previously initiated procedure without relinquishing control of the call:

```
SGNLPROCEDURE (tid,  yourtid, parameters)
               INDEX BOOLEAN  LIST
```

Like the TAKEPROCEDURE procedure already described, its arguments include the identifier TID of the affected transaction, an indication YOURTID of from whose name space the identifier was assigned, and the PARAMETERS themselves.

This new procedure enables the environment to make available to its applications program a primitive that has a calling sequence similar to that of the system procedure but which does not require YOURTID as an argument. Its implementation requires only that the environment identify the remote process via its internal tables and invoke SGNLPROCEDURE in that process.

By requesting the acknowledgment of each call to SGNLPROCEDURE and, if necessary, delaying subsequents calls affecting the same transaction until the acknowledgment arrives, the invoking environment effects a crude form of flow control and so prevents the remote process' buffers from being overrun.

6.3 Soliciting Help from Superiors 4f4

As in conventional programming systems, remotely callable procedures within a distributed system will sometimes call upon others to carry out portions of their task. Each procedure along the "thread of control" resulting from such nested calls is, in a sense, responsible to not only its immediate caller but also to all those procedures that lie above it along the control thread. To properly discharge its responsibilities, a procedure must sometimes communicate with these "superiors".

Occasionally a procedure reaches a point in its execution beyond which it cannot proceed without external assistance. It might, for example, require additional resources or further direction from the human user upon whose behalf it is executing. Before reaching this impasse, the procedure may have invested considerable real and/or processing time that will be lost if it aborts.

Adding to the Protocol the following system procedure minimizes such inefficiencies by enabling the environment to solicit help from a callee's superiors:

```
HELPPROCEDURE (tid,  number, information -> solution)
               INDEX INDEX   any            any
```

Its arguments include the identifier TID of the affected transaction (the direction of the control transfer being implicit in this case), a NUMBER identifying the problem encountered, and arbitrary supplementary INFORMATION.

The primitive that this new procedure enables the environment to provide its applications program has an identical calling sequence. Its implementation requires only that the environment identify the remote process from its internal tables and invoke HELPPROCEDURE in that process.

The search for help begins with invocation of HELPPROCEDURE in the caller's environment. If the caller understands the problem

(that is, recognizes its number) and is able to solve it, HELPPROCEDURE will simply return whatever SOLUTION information the caller provides. Otherwise, HELPPROCEDURE must give the next superior an opportunity to respond by calling itself recursively in that process. The search terminates as soon as a superior responds positively or when the end of the control thread is reached. In the latter case, each of the nested HELPPROCEDURE procedures returns unsuccessfully to indicate to its caller that the search failed.

6.4 Reporting an Event to Superiors 4f5

A procedure sometimes witnesses or causes an event of which its superiors should be made aware (for example, the start or completion of some major step in the procedure’s execution). Adding to the Protocol the following system procedure enables the environment to notify a callee’s superiors of an arbitrary event:

```
NOTEPROCEDURE (tid,  number, information)
               INDEX INDEX    any
```

Like HELPPROCEDURE, its arguments include the identifier TID of the transaction it affects, a NUMBER identifying the event being reported, and arbitrary supplementary INFORMATION.

The primitive that this new procedure enables the environment to provide its applications program has an identical calling sequence. Its implementation requires only that the environment identify the remote process from its internal tables and invoke NOTEPROCEDURE in that process.

By requesting acknowledgment of each call to NOTEPROCEDURE and, if necessary, delaying subsequents calls that affect that transaction until the acknowledgment arrives, the invoking environment effects a crude form of flow control and so prevents the remote process’ buffers from being overrun.

Notification of the procedure’s superiors begins with invocation of NOTEPROCEDURE in the caller’s process and works its way recursively up the thread of control until the top is reached.

7. Aborting Executing Procedures 4g

Conventional systems that accept commands from the user sometimes permit him to cancel an executing command issued inadvertently or with erroneous parameters, or one for whose completion he cannot wait. This ability is particularly important when the command (for example, one that compiles a source file) has a significant execution time. In a distributed system, the execution of such a command may involve the invocation of one or more remote procedures. Its cancellation, therefore, requires the abortion of any outstanding remote procedure calls. 4g1

Adding to the Protocol the following system procedure provides the basis for a command cancellation facility by enabling the environment to abort another, previously invoked procedure: 4g2

```
   ABRTPROCEDURE (tid)
                  INDEX
```

Its sole argument is the identifier TID of the transaction it affects. 4g3

The primitive that this new procedure enables the environment to make available to the applications program has an identical calling sequence. Its implementation requires only that the local environment identify the remote process from its internal tables and invoke ABRTPROCEDURE in that process. 4g4

CONCLUSION 5

The expanded Protocol and Model that result from the extensions proposed in the present paper are summarized in Appendixes B and C, respectively. Needless to say, many additional forms and aspects of process interaction, of which Appendix D suggests a few, remain to be explored. Nevertheless, the primitives already made available by the run-time environment provide the applications programmer with a powerful and coherent set of tools for constructing distributed systems. 5a

ACKNOWLEDGMENTS 6

Many individuals within both SRI's Augmentation Research Center (ARC) and the larger ARPANET community have contributed their time and ideas to the development of the Protocol and Model described in this and its companion paper. The contributions of the following individuals are expressly acknowledged: Dick Watson, Jon Postel, Charles Irby, Ken Victor, Dave Maynard, and Larry Garlick of ARC; and Bob Thomas and Rick Schantz of Bolt, Beranek and Newman, Inc. 6a

ARC has been working toward a high-level framework for network-based distributed systems for a number of years now [2]. The particular Protocol and Model result from research begun by ARC in July of 1974. This research included developing the Model; designing and documenting the Protocol required to support it [3]; and designing, documenting, and implementing a prototype run-time environment for a particular machine [4, 5], specifically a PDP-10 running the Tenex operating system developed by Bolt, Beranek and Newman, Inc [6]. Three design iterations were carried out during a 12-month period and the resulting specification implemented for Tenex. The Tenex RTE provides a superset of the capabilities proposed in this paper. 6b

The work reported here was supported by the Advanced Research Projects Agency of the Department of Defense, and by the Rome Air Development Center of the Air Force. 6c

APPENDIX A: TRANSMISSION FORMATS FOR PCP DATA OBJECTS 7

Data objects must be encoded in a standard transmission format before they can be sent from one process to another via the Protocol. An effective strategy is to define several formats and select the most appropriate one at run-time, adding to the Protocol a mechanism for format negotiation. Format negotiation would be another responsibility of the environment and could thus be made completely invisible to the applications program. 7a

Suggested below are two transmission formats. The first is a 36-bit binary format for use between 36-bit machines, the second an 8-bit binary, "universal" format for use between dissimilar machines. Data objects are fully typed in each format to enable the environment to automatically decode and internalize incoming parameters should it be desired to provide this service to the applications program. 7b

PCPB36, For Use Between 36-Bit Machines 7c

Bits 0-13 Unused (zero) 7c1

Bits 14-17 Data type 7c2

EMPTY =1 INTEGER=4 LIST=7
BOOLEAN=2 BITSTR =5
INDEX =3 CHARSTR=6

Bits 18-20 Unused (zero) 7c3

Bits 21-35 Value or length N 7c4

EMPTY	unused (zero)
BOOLEAN	14 zero-bits + 1-bit value (TRUE=1/FALSE=0)
INDEX	unsigned value
INTEGER	unused (zero)
BITSTR	unsigned bit count N
CHARSTR	unsigned character count N
LIST	unsigned element count N

Bits 36- Value 7c5

EMPTY	unused (nonexistent)
BOOLEAN	unused (nonexistent)
INDEX	unused (nonexistent)
INTEGER	two's complement full-word value
BITSTR	bit string + zero padding to word boundary
CHARSTR	ASCII string + zero padding to word boundary
LIST	element data objects

PCPB8, For Use Between Dissimilar Machines 7d

```
   Byte    0  Data type                                            7d1
      EMPTY  =1  INTEGER=4  LIST=7
      BOOLEAN=2  BITSTR =5
      INDEX  =3  CHARSTR=6
   Bytes 1-   Value                                                7d2
      EMPTY      unused (nonexistent)
      BOOLEAN    7 zero-bits + 1-bit value (TRUE=1/FALSE=0)
      INDEX      2-byte unsigned value
      INTEGER    4-byte two's complement value
      BITSTR     2-byte unsigned bit count N + bit string
                   + zero padding to byte boundary
      CHARSTR    2-byte unsigned character count N + ASCII string
      LIST       2-byte element count N + element data objects
```

APPENDIX B: THE EXPANDED PROCEDURE CALL PROTOCOL 8

The Protocol that results from the extensions proposed in this paper is summarized below. The reader should note the concise syntactic description made possible by the underlying notion of PCP data types. 8a

Parameter list masks have been included not only as additional parameters of the CALL message, as proposed in the paper, but as arguments of the TAKEPROCEDURE and SGNLPROCEDURE system procedures as well. Throughout the Protocol description, "MASK" is shorthand for: 8b

```
   [LIST (variable [CHARSTR], ...)]                                  8b1
```

Messages 8c

```
   LIST (route INDEX, opcode INDEX CALL=1, tid [INDEX],
         pkh [INDEX], procedure CHARSTR, arguments LIST,
         argumentlistmask MASK, resultlistmask MASK)                 8c1
   LIST (route INDEX, opcode INDEX RETURN=2, tid INDEX,
         outcome BOOLEAN, results LIST)                              8c2

      If OUTCOME is FALSE,
         RESULTS is LIST (error INDEX, diagnostic CHARSTR)
```

Process-Related System Procedures 8d

```
   INIPROCESS (program CHARSTR,
               credentials LIST (user CHARSTR, password CHARSTR,
                                 account CHARSTR))                   8d1
   ALOPORT    (-> ph INDEX, computer CHARSTR, port)                  8d2
   CNNPORT    (ph INDEX, computer CHARSTR, port)                     8d3
   DCNPORT    (ph INDEX)                                             8d4
   CRTROUTE   (mycode INDEX, oldcode [INDEX]
               -> code INDEX, ph INDEX)                              8d5
   DELROUTE   (yourcode INDEX)                                       8d6
```

Package-Related System Procedures 8e

```
   OPNPACKAGE (packages LISTofCHARSTRs -> pkhs LISTofINDEXs)        8e1
   CLSPACKAGE (pkhs LISTofINDEXs)                                    8e2
```

```
Variable-Related System Procedures                                  8f

   CRTVARIABLE (variable CHARSTR, value)                            8f1
   DELVARIABLE (variable CHARSTR)                                   8f2
   RDVARIABLE  (pkh INDEX, variable CHARSTR,
                substructure [LISTofINDEXs] -> value)               8f3
   WRVARIABLE  (pkh INDEX, variable CHARSTR,
                substructure [LISTofINDEXs], value)                 8f4

Procedure-Related System Procedures                                 8g

   TAKEPROCEDURE (tid INDEX, yourtid BOOLEAN, parameters LIST,
                  argumentlistmask MASK, resultlistmask MASK)       8g1
   SGNLPROCEDURE (tid INDEX, yourtid BOOLEAN, parameters LIST,
                  parameterlistmask MASK)                           8g2
   HELPPROCEDURE (tid INDEX, number INDEX, information
                  -> solution)                                      8g3
   NOTEPROCEDURE (tid INDEX, number INDEX, information)             8g4
   ABRTPROCEDURE (tid INDEX)                                        8g5
```

APPENDIX C: SUMMARY OF RTE PRIMITIVES 9

The DPS primitives made available to the applications program as a result of the Model extensions proposed in this paper are summarized below. Collectively, they provide the applications programmer with a powerful and coherent set of tools for constructing distributed systems. Some of the primitives (for example, CRTPROCESS and DELPROCESS) are necessary elements of a "network operating system (NOS)", into which DPS may itself one day evolve. 9a

Processes 9b

```
   CRTPROCESS (computer, program, credentials -> ph)                9b1
   DELPROCESS (ph)                                                  9b2
   ITDPROCESS (ph1, ph2 -> ph12, ph21, ih)                          9b3
   SEPPROCESS (ih)                                                  9b4
```

Packages 9c

```
   OPNPACKAGE (ph, packages -> pkhs)                                9c1
   CLSPACKAGE (ph, pkhs)                                            9c2
```

Variables 9d

```
   CRTVARIABLE (ph, variable, value)                                9d1
   DELVARIABLE (ph, variable)                                       9d2
   RDVARIABLE  (ph, pkh, variable, substructure -> value)           9d3
   WRTVARIABLE (ph, pkh, variable, substructure, value)             9d4
```

Procedures 9e

```
   CALLPROCEDURE (ph, pkh, procedure, arguments, argumentlistmask,
                  resultlistmask, -> outcome, results, tid)         9e1
   LINKPROCEDURE (tid, arguments, argumentlistmask,
                  resultlistmask, -> outcome, results)              9e2
   SGNLPROCEDURE (tid, parameters, parameterlistmask)               9e3
   HELPPROCEDURE (tid, number, information -> solution)             9e4
   NOTEPROCEDURE (tid, number, information)                         9e5
   ABRTPROCEDURE (tid)                                              9e6
```

APPENDIX D: ADDITIONAL AREAS FOR INVESTIGATION 10

Although the expanded distributed programming system developed in this paper and summarized in the previous appendix is already very powerful, many additional aspects of process interaction remain, of course, to be explored. Among the additional facilities that the Protocol must eventually enable the environment to provide are mechanisms for: 10a

(1) Queuing procedure calls for long periods of time (for example, days). 10a1

(2) Broadcasting requests to groups of processes. 10a2

(3) Subcontracting work to other processes (without remaining a middleman). 10a3

(4) Supporting brief or infrequent inter-process exchanges with minimal startup overhead. 10a4

(5) Recovering from and restarting after system errors. 10a5

REFERENCES 11

1. White, J. E., "A High-Level Framework For Network-Based Resource Sharing," Submitted for publication in the AFIPS Conference Proceedings of the 1976 National Computer Conference. 11a

2. Watson, R. W., Some Thoughts on System Design to Facilitate Resource Sharing, ARPA Network Working Group Request for Comments 592, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, November 20, 1973 (SRI-ARC Catalog Item 20391). 11b

3. White, J. E., DPS-10 Version 2.5 Implementer's Guide, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, August 15, 1975 (SRI-ARC Catalog Item 26282). 11c

4. White, J. E., DPS-10 Version 2.5 Programmer's Guide, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, August 13, 1975 (SRI-ARC Catalog Item 26271). 11d

5. White, J. E., DPS-10 Version 2.5 Source Code, Augmentation Research Center, Stanford Research Institute, Menlo Park, California, August 13, 1975 (SRI-ARC Catalog Item 26267). 11e

6. Bobrow, D. G., Burchfiel, J. D., Murphy, D. L., Tomlinson, R. S., "TENEX, a Paged Time Sharing System for the PDP-10," Communications of the ACM, Vol. 15, No. 3, pp. 135-143, March 1972. 11f

FIGURE LIST 12

Figure 1. Interfacing distant applications programs via their run-time environments and an IPC channel. 12a

Figure 2. Two processes that can only be introduced via a logical channel. 12b

Figure 3. A logical channel. 12c